AOC2 2nd Course Project - Data Memory Hierarchy

AOC2 Team, EINA, Univ. Zaragoza

Submission deadline: 12 de mayo a las 9:00

README

- **Save time by carefully reading this Guide.**
  - You will understand key issues still possibly unclear to you despite class theory and problems
  - It will help you organize and write your report.
- **Do not change the name of the components and signals** used in the provided VHDL source code without checking it out with us.
- All materials (source files, slides, problems, guides...) are under the Intellectual Property Rights of the University of Zaragoza. **The violation of these rights can entail legal consequences**.
- **Language issues** - AOC2 has been declared an ELF (*English Language Friendly*) course. The main reference books and most in-house written materials, are delivered in English. We are still transitioning from a non-ELF course, and therefore you will find symbols and comments in Spanish throughout the provided VHDL source code. Do not hesitate to ask the faculty if you are a non-native Spanish speaker and run into trouble with this.
- Report any typo or mistake you may find and ask about anything you need, but mind that we will drop questions which are already answered in this Guide.

☐ The check-boxes throughout this Guide constitute the checklist you must complete before submitting your project.

Contents

1 Summary

In this project, you will work with a more complex memory hierarchy than in the first one. The components communicate over a semi-synchronous bus and comprise a cache memory (`MC`), data memory (`MD`), and scratch data memory (`MD Scratch`). The `MD` provides the same support for the `lwi` instruction as in the first project, but operates at a lower speed than the `MD Scratch`, whose contents are *non-cacheable* (i.e. they are never be stored in `MC`). Data loaded via `lwi` is also non-cacheable. Our `MD` memory supports `lwi` operations, but you have to identify whether `MD Scratch` also supports it or not. In the latter case `lwi` must be handled as a regular `lw`. The assignment consists in designing a cache memory controller that handles processor requests and manages data transfers across the bus accordingly. The `MC` includes additional performance counters. Presentation and defense will mirror the procedures of the first project. Please carefully read this guide, its annexes, and the submission instructions we will provide in due time.

2 Baseline system

In this second project, the memory system encompasses a cache memory (`MC`), the data memory (`MD`), a *scratch* memory (`MD Scratch`), a semi-synchronous bus, and an I/O peripheral (`IO_Master`). Both the `MC` controller and the `IO_Master` controller are bus managers[1]. Therefore, the bus requires an arbiter to avoid conflicts. To ensure fair bus access, the arbiter uses dynamic priority assignment. The interface with the MIPS is identical to that of the memory in the first project. Fig. 1 shows the bus signal diagram described below.

2.1 SOC

You should use the SoC modules of the first project as is but for the IO_MD_subsytem, which changes completely. Just keep the same modules for the MIPS and SoC we provided along with the four components you submitted, replace the old `IO_MD_subsystem.vhd` file with new one, and add all its auxiliary components.

[1] We follow the recommendations in 2021 IEEE SA Standards Style Manual on inclusive language, avoiding master/slave in favor of descriptive terms.

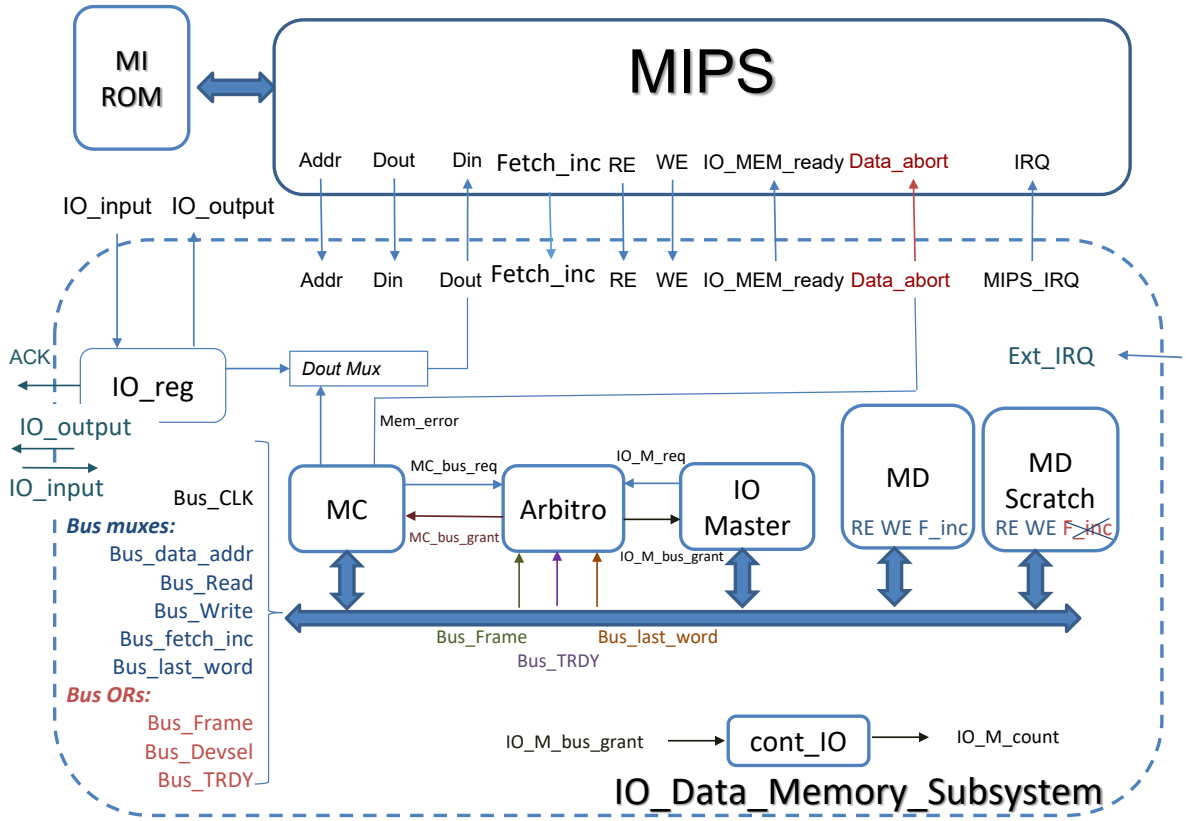

Figure 1: Principal components and signals in the `IO_MD_subsytem` of this project. The assignment consist in designing the `MC` Control Unit

.

2.2 Memory bus

This is a semi-synchronous bus loosely based on the PCI standard, where `MC` and `IO_Master` act as bus managers, and `MD` and `MD Scratch` always act as clients. **Data and address lines are multiplexed**. The bus supports variable-size bursts. Transfers consist of three phases:

- **Arbitration** – Before starting a transfer, the bus managers must request the bus using the `Bus_req` signal. The arbiter will respond by asserting `Bus_grant`, though there may be a wait if the bus is busy, since there are two managers.
- **Address** – **The transfer begins at the next cycle after the assertion of** `Bus_grant`. The manager asserts the `Bus_Frame` signal and sends the address over the `Bus_data_addr` lines along with the operation type (`Bus_Rd_Wr` = '0' for reads, '1' for writes). **The client that recognizes the address as belonging to its addressing space will respond by asserting** `Bus_DevSel` **in the same cycle**.
  - **Important:** If no device asserts `devsel` in the cycle when the address is sent, the requested address must be stored in a register (`ADDR_Error_Reg`), and the error bit `MEM_Error` must be set. The signal `MEM_Error` will trigger a `Data_abort` in the MIPS. The `ADDR_Error_Reg` register is readable from the MIPS at address `x"01000000"`; when the processor reads it, the error bit must be cleared.
  - There are two other sources of memory error (i.e., triggers for `MEM_Error`): unaligned access (as in the previous project), and any attempt to write to the `ADDR_Error_Reg` register, which is read-only.
- **Data** – This phase begins **right in the cycle following the assertion of** `Bus_DevSel` by the target. During this phase, the manager holds `Bus_Frame`, and the target: a) Signals it is ready to

perform the requested operation by asserting `Bus_TRDY` (*target ready*) in the current cycle; and b) Sends or receives one data word per cycle via `Bus_Data_Addr` until the manager deasserts `Bus_Frame`. At that moment, the target must remove the data from the bus. Therefore, `Bus_Frame` must not be deasserted prematurely. If the target is not able to provide the expected data in a given cycle, **it will deassert `Bus_TRDY` to indicate the manager must wait**. Data is always sent/received consecutively on the bus. For example, if the manager requests word 4, the target will send word 4, then 8, then 12, and so on. **When the manager sends or requests the last word, it will assert the `last_word` signal**. If only one word is requested, `last_word` will also be asserted, since it is both the first and last word.

2.3 MC

The `MC` is a two-way (s=2) set-associative cache with 128 bytes (32 words), FIFO replacement, write-through update policy, and conventional allocate-fetch policy on write miss. Each way holds 4 blocks of 4 words.

2.4 MD Scratch

The `MD Scratch` is a small, fast data memory. Its access latency is as low as the `MC`, but it is explicitly managed through `lw` and `sw` instructions within the address range `X"10000000"`–`X"100000FF"`. `lw` and `sw` to these addresses do not cause any faults or data exchanges with `MD`. Study the code in `MD_scratch.vhd`, find if the MIPS can issue a `lwi` to the `MD Scratch` or not. If not, `lwi` must be handled as a regular `lw`. Addresses in the `MD Scratch` are *non-cacheable*: the `MC` controller will forward to the bus any read or write request coming from the MIPS and targeting the `MD Scratch` addressing space, **but no block will be fetched** into the `MC`. Following an `lw` to `MD Scratch`, the `MC` controller forwards the word returned by `MD Scratch` to the MIPS. Burst mode is not supported. That is, words must be accessed one at a time.[2]

2.5 MC controller

The `MC` controller receives requests from the MIPS and, if possible, responds with data stored in `MC` or in the I/O registers. Otherwise, it performs the necessary transfers with `MD` or `MD Scratch` via the bus. Alongside the project materials on Moodle, you will find a full-page schematic of the internal organization of `MC`, which serves as a reference for designing the controller. **The controller is a FSM** (finite state machine) of moderate complexity. The challenge lies in fully understanding the protocols and their signals.

Caution: `lwi` can generate a cache coherence problem that must be solved by invalidating the `MC` memory block. Think of a `lw` miss, which makes the `MC` controller load the missing block into the `MC`. Then, a `lwi` references the same address, or any other address in that block. The `MC` controller just forwards the request to the `MD` through the bus, ignoring the tag lookup. Therefore, `MD` increments and internally updates the word requested by `lwi`, returns the original value through the bus, and the `MC` controller forwards it to the MIPS, without updating it in the `MC` block. A potential ulterior `lw` requesting that word again could hit the `MC` if the block has not been evicted yet, reading the outdated value stored in `MC` instead of the updated value in `MD`. A `sw` hit will even override the value in `MD` previously increased by the `lwi`. Hence, the `MC` controller must identify addresses hitting in `MC` and issued by a `lwi`, and invalidate the corresponding block asserting the `invalidate_bit` signal.

The `MC` controller includes a **second FSM** with two states (`Memory_error` and `No_error`), that you must also implement either as part of the main FSM process, due to their relation, or separately. This FSM is responsible for generating the `Mem_Error` signal, which the MIPS interprets as a `Data_abort`. The controller transitions from state `No_error` to state `Memory_error` when: a) No device acknowledges the sent address; b) The MIPS requests a misaligned word; or c) The MIPS attempts to write to a read-only register. In this state, the controller continues normal operation, but keeps the `Mem_Error` signal asserted. It also **stores the address (or byte) of the word that triggered the error in the `Addr_Error_Reg` register**. The controller **remains in the error state until the MIPS reads the `Addr_Error_Reg` register**, which is memory-mapped at address `x"01000000"`. Once the MIPS reads this register, the controller returns to the `No_error` state.

[2] Early GPGPUs lacked cache memory but included this type of *scratch* memory. In NVIDIA/CUDA it is called *shared memory*, and in OpenCL, *scratchpad*. Starting with NVIDIA's Fermi architecture, a level was introduced that can be configured partly as cache, partly as *scratchpad*, at the programmer's discretion. *Scratchpads* are also common in SoCs for embedded systems. Our `MD Scratch` is inspired by this concept.

2.6 MD controller

The MD controller always acts as a target, forwarding bus requests to MD if they fall within its address space (X"00000000"–X"000001FF"). Addresses outside this range are ignored.

Timing: This is just MD the same MD of the first project, but with added delays: the latency for the first word (L) is higher than for subsequent words (R)[3]. When the MD cannot provide data in the current cycle, it will deassert the Bus_TRDY signal to indicate that the controller must wait.

Burst mode: When Bus_Frame is asserted, the data memory will perform the requested operation with the initial address and then increment it internally.

2.7 Event Counters in MC

The MC includes **four counters**[4]:

1. **cont_m** Misses in MC. A miss is defined as an access (either lw or sw) to a **cacheable address** absent in MC. Accesses to MD Scratch or to the internal registers of MC are ignored as misses.
2. **cont_w** Write operations to MC. Writes to I/O registers or MD Scratch are ignored.
3. **cont_r** Read operations from MC. Reads from I/O registers, or MD Scratch, or lwi are ignored.
4. **cont_inv**: It increments every time that a block stored in MC is invalidated due to a lwi.

Caution: Mind that each event must only increment once the relevant counters. E.g., if a miss takes 23 cycles to handle, cont_m increments only once, not 23 times. Track how many reads, writes, misses, and invalidations occur in your testbench, and verify that the values you track in the testbench match the values of the MC counters you see in your waveform. These counters are very useful for navigating the waveform during debugging.

2.8 IO_Master

The system includes an input/output peripheral (IO_Master) acting as a bus manager. It monitors the system input IO_input on every cycle and continuously attempts to access the bus to write the input value into the first word of MD Scratch. The counter cont_IO is incremented each cycle in which the IO_Master successfully gains bus access. The more efficient the transfer management by your MC controller is, the more bus cycles will be available to IO_Master—i.e., the higher the value of cont_IO.

3 Results and deliverables

3.1 Tests

You must first check that your design works correctly for all possible cases read and write hits and misses using all four sets, lwi, and invalidation due to lwi, and that the counters increment when appropriate. You must also perform reads and writes to MD Scratch, generate a memory error, verify that the abort signal is raised, and confirm that it clears when the internal MC register is read. To do this, we recommend the following testing methodology.

3.1.1 Unitary tests

Leverage a testbench to test and spot specific, isolated scenarios. We provide a *testbench* example that tests read misses, read hits, and clean replacements across the entire MC. Try it on your design and gradually add other scenarios: write hits and misses, dirty replacements, reads and writes to MD Scratch, I/O registers, internal MC registers, and abort handling. The source files include a *testbench* without IRQs to simplify debugging.

3.1.2 Integration tests

Write an assembly program encompassing the cases you tested in your testbench. It can reveal side effects which might not be obvious when testing with a series of shorter program tests. Then you can comment on this code in your project report as you did in the first project.

[3] Recall the baseline bus model explained in class

[4] Real controllers often include more counters to evaluate performance and help adapt programs to the memory hierarchy.

3.2 Project report

Start with a short summary ($\tilde{1}00$ words) briefing on your design. Avoid long textual descriptions and leverage code/pseudocode snippets and diagrams throughout the report. The following items are mandatory:

1. **State graph of the Control Unit**, explaining the operations performed in each state and the signals activated. You can use a table with a per-state and /or per-transition entry to get into detail while keeping the graph simpler. Avoid long explanations: focus on which signals change and why, especially the key ones.
2. A figure showing the **address breakdown** for addressing the `MC` as we do in class (*byte/word – set – tag*).
3. **Latency analysis of the different bus transfers** excluding arbitration-induced delays. Obtain the following latencies from the simulation:
   - `CrB(MD)`: Cycles required to read a block from `MD`, where `CrB(MD)` = L cycles (first word latency) + $3 \times R$ cycles (latency for the remaining words).
   - `CwW(MD)`: Cycles required to write a word to `MD`.
   - `Clwi(MD)`: Cycles required to carry out a `lwi` to `MD`.
   - `CrW(MDscratch)`: Cycles required to read a word from `MD Scratch`.
   - `CwW(MDscratch)`: Cycles required to write a word to `MD Scratch`.
   - `CrW or CwW(IO_REG)`: Cycles required to read from or write to internal I/O registers.
4. **Mathematical formulation to compute the average cycles per memory access** as a function of the costs given above, and of the possible events taking place at your `MC`: read/write hit, read/write miss to `MD` or `MD Scratch`, `lwi`, or access to the internal register of `MC`. If unsure about specific values, verify via simulation. Note that **the arbitration latency is not constant in this system**: assume an average of 1.5 cycles for calculations.
5. **Unitary tests**: testbench or program snippets, properly commented, testing specific events. For example, a read miss followed by a stream of hits, read misses on the same set triggering a replacement, a read miss followed by a write hit w/ and w/o replacement, references to the `MD Scratch`, `lwi` to the `MD Scratch` ... or whatever cases you consider important to prove that your control unit works fine. Identify all states in your state machine and **present a table specifying where and how each case has been verified**.
6. **Integration test: code example**. Assembly program leveraging the memory hierarchy. Compute the ***speedup*** provided by `MC` and `MD Scratch` compared to a similar system that only includes the `MD` of this project (not from the 1st. Project Assignment). **Include in the test at least one invalidation**. You may want to reuse one of your test programs if suitable to highlight the impact of the `MC`. Speedup can be calculated theoretically or upon cycle counts obtained from simulations (w/ and w/o the cache).
7. Report the number of hours each group member dedicated to each part of the project, along with the **total number of hours**.
8. A **self-assessment**, to be completed individually, answering two questions:
   - Do you believe you achieved the course objectives?
   - What grade would you give yourself?

4 Grading policy

Remember that we individually assess each submitted assignment using our own test program. As in the 1st Project Assignment, **the design must perform correctly in order to pass**. If we detect minor errors, we will provide our testbench with step-by-step annotated source code and pre-loaded memory files. You will have 24 hours to fix your errors and re-submit.

The state graph must be clear and easy to follow. Instead of including all signals in the diagram, you can resort to a table to detail the behavior of each state (Moore) or transition (Mealy) in the text, clearly identifying the state number or transition. **Your tests must specify whether each `lw`, `lwi` or `sw` results in a hit or miss. Indicate set and way for cached data. If this analysis is incorrect or inexistent, we will not even verify your design**.

Failure to clearly explain signal value assignments, particularly those at the `MC` control unit, either in your report or during the interview, may also lead to a failing grade. Answers such as *. . . if I put '0' instead of '1' it doesn't work* are not valid. This is not a warning meant to intimidate: 99% of you will have no issues. We just want to make it clear that submitting a solution without demonstrating understanding is not acceptable. This also applies to re-submissions, which we review thoroughly. The appearance of even minor bugs can prompt deeper inspection of your code, and you will fail the assignment if we find any inconsistencies.

Grading criteria: The main evaluation criteria will be whether the **state diagram is correct, clear, and well explained**, and whether the simulation is coherent with your description and formula for the average memory access cycles. Less critically, we will also consider whether the controller is efficient and minimizes stall cycles. Start with a straightforward design, and optimize it later if time permits (see options below).

5 Optional extensions

- *A basic lock-up free cache.*- Include a ***buffer* for writes to `MD`**. When a write to `MD` is required, the data and its address are stored in dedicated registers, and the processor is signaled to continue by asserting `Mem_Ready`. The `MC` may continue handling hits while the Control Unit manages the write over the bus. Execution stalls if a second bus transfer is needed before completing the first. Identify a few cases where this mechanism improves performance.

- *Ethical hacking.*- Implement and evaluate a function that degrades performance by saturating `MC` with dirty blocks, triggering evictions on every miss. Assess its detectability in our SoC and possible countermeasures.

- *Critical word forwarding.*- On a read miss, the `MC` controller fetches the full block and then sends the requested word to the CPU. Performance can be improved by forwarding the requested word as soon as it arrives, allowing the CPU to proceed while the controller finishes retrieving the rest of the block. Describe scenarios where this technique could enhance performance.